Real-time 3D crying simulation

INF/SCR-08-73

Author:
Wijnand van Tol
3084566

Supervisor:
Dr. ir. A. Egges

June 17, 2009
Abstract

Displaying facial motions such as crying or laughing is difficult to achieve in real-time simulations and games. Not only because of the complicated simulation of the physical characteristics such as muscle motions or fluid simulations, but also because one needs to know how to control these motions on a higher level. In this thesis, we propose a method that is an extension of the MPEG-4 Facial Animation standard to control a realistic crying face in real-time. The tear simulation is based on the Smoothed Particle Hydrodynamics technique, which we optimised for real-time tear generation and control. Through simple parameters, a wide range of expressions and tears can be generated on the fly. Additionally, our method works independently of the graphics and physics engines that are used.
# Contents

1 Introduction 3

2 Related work 6
   2.1 Introduction ............................................. 6
   2.2 The Psychology of Crying .................................. 6
   2.3 Facial Animation .......................................... 7
   2.4 Tear simulation ........................................... 9
      2.4.1 Fluid Simulation ..................................... 10
      2.4.2 Fluid Visualisation ..................................... 10
   2.5 Motivation and Goal ..................................... 13

3 Facial Animation 14
   3.1 Introduction ............................................. 14
   3.2 Facial Animation .......................................... 14
   3.3 Implementation ........................................... 19
   3.4 Summary .................................................. 21

4 Fluid Simulation 22
   4.1 Introduction ............................................. 22
   4.2 Smoothed Particle Hydrodynamics......................... 22
   4.3 Visualisation ............................................. 27
      4.3.1 Fluid Visualisation ..................................... 27
      4.3.2 Fluid Trail Visualisation ............................. 30
   4.4 Implementation ........................................... 32
      4.4.1 Smoothed Particle Hydrodynamics ..................... 32
      4.4.2 Data structure ........................................ 33
      4.4.3 Marching Cubes ....................................... 36
   4.5 Summary .................................................. 38

5 Integration 39
   5.1 Introduction ............................................. 39
   5.2 Tear generation .......................................... 39
   5.3 Texture Blending ......................................... 40
   5.4 Physical Parameters ...................................... 40
   5.5 User Interface ........................................... 41
   5.6 Summary .................................................. 43
6 Results

6.1 Introduction ........................................... 44
6.2 General Performance .................................. 44
6.3 Pressure .................................................. 45
6.4 Viscosity ............................................... 46
6.5 Surface Tension ........................................ 47
6.6 Adhesion ............................................... 47
6.7 Particle Count ......................................... 47
6.8 Trail Synthesis ........................................ 49
6.9 Texture Blending ...................................... 50
6.10 Frame Rate Measurements ............................ 50
6.11 Summary .............................................. 51

7 Conclusions and Future Work ......................... 52

7.1 Introduction ........................................... 52
7.2 Conclusions .......................................... 52
7.3 Future Work .......................................... 53

A FAP Definition Table .................................. 57

B Simulation Parameters ................................. 61
Chapter 1

Introduction

Virtual characters in games and simulations are becoming more realistic, not only through more detailed geometry but also because of better animations. Due to motion capturing and automatic blending methods, both facial and body motions can be displayed convincingly. One of the main goals of using virtual characters in 3D environments is to make an affective connection with the user. An important aspect of establishing this connection between the virtual character and the user is the expression of emotions. Virtual characters are already capable of displaying a variety of emotions, either by employing different body motion styles or facial expressions. An example of a genre where interaction with characters is important is in Role Playing Games. For example, a game which uses emotions to enhance conversations with computer controlled characters is Oblivion: Elder Scrolls IV [29]. In this game, there are four emotions that a character can have, but these are expressed only through the use of very static expressions. An example of this is shown in Figure 1.1.

![Four expressions used in the game Oblivion: Elder Scrolls IV [29]. They are very static, but they are influenced by the user, which increases immersion.](image)

In newer games, such as Fallout 3 [30] or Grand Theft Auto IV [31], characters display a wider variety of emotions, but these emotions seem hardly affected by the actions of the user. For example, the friendly characters stay friendly, and always look friendly, while unfriendly characters always look angry. Figure 1.2 shows a character from Fallout 3, where the facial expressions vary slightly, but these expressions are not dependent on user input.
In the previous examples, emotions are solely shown by animating the geometry of the face. However, extreme emotions such as laughing out loud, screaming in anger or crying require more than displaying expressions. For example, when someone is screaming, this person’s face might get red or when someone is crying, tears roll down the face of that person. The absence of these extreme emotions results in a less emotionally involving experience, as opposed to movies, where such expressions are used regularly to entice the viewer. A technical demonstration of the game Heavy Rain [32] showed the first crying character in real-time 3D. A screenshot from this movie can be seen in Figure 1.3.

The expressions shown in this real-time cinematic look almost life-like, but are directly taken from motion capture. The crying is very realistic and expresses the anger that the woman in this movie is feeling, but they follow a scripted event that the viewer has no effect on. For a player or viewer to have control over the emotions of a character, crying in particular, we should be able to generate a crying expression, with tears, on the fly, by using parameters which determine the mood of the character. Displaying these extreme expressions in real-time 3D is a complicated task. It requires precise muscle modelling and skin deformation. In many applications, facial animation is done by a geometrical deformation approach, usually in combination with a facial animation standard such as the MPEG-4 Facial Animation standard. This standard for facial animation defines the key feature points of a face and their movement, allowing an animator to describe and recreate virtually every expression by using a limited set of parameters. The standard only defines deformations of the face and does not account for other emotional effects such as tears as a result of crying. Generating tears requires real-time fluid simulation that realistically interacts with the face. Because of the technical challenges, few games try to incorporate extreme emotions, and if they do, it is mostly done by using manually defined motions. In this thesis, we propose a system for automatically generating and displaying crying motions in real-time. We achieve this by using a real-time fluid simulation method, which we optimised for crying synthesis. We will also show that with a simple extension of the MPEG-4 Facial Animation standard, the real-time crying engine can be controlled by an animator, without having any knowledge of the physical processes that govern the crying simulation.
Figure 1.3: A screenshot of the Heavy Rain [32] technical demonstration. The crying is very realistic but it is preprogrammed and not influenced by the user.
Chapter 2

Related work

2.1 Introduction

In this chapter, we will give an overview of research that is related to crying simulation. First, we need an understanding on the psychological aspect of crying. For this purpose, we will discuss research that explains why people cry and what kind of phenomena are mostly associated with crying. Second, we will discuss earlier methods of facial animation, because facial expressions are a part of effectively simulating crying. Therefore, we need to inventorise the available methods and standards if we want to create a successful facial animation engine. Finally, we will look into related work on fluid simulation, needed for simulating tears. Tears are an example of an aspect of crying that is not associated with expressions and geometrical deformation of the face.

2.2 The Psychology of Crying

In psychology, research has been done on why people cry. For us, this research is important if we want to identify the parameters of a situation in which someone cries. Being able to describe the circumstances of someone who is crying can be used for several things. The current emotion of a person, for example, provides a lot of information about their current facial expression. Also, reproducing a certain situation through the use of parameters can be used as a more elaborate control mechanism for crying. There have been several studies that investigate why people cry, and these have been summarised by Vingerhoets et al. [24]. In particular, we would like to mention Borgquist [4] who did a study among students, which pointed out three mood states in which crying occurs: anger, grief or sadness, and joy. He also pointed out accompanying physical states such as fatigue, stress and pain. Young [26] studied crying on college students, but made no clear distinction between moods and situations. He classified the reasons for crying as follows: disappointment, lowered self esteem, unhappy mood, organic state, special events and laughter to the point of tears. William and Morris [25] proposed a list of possible crying inducing events or moods, containing situations such as deaths of intimates, broken love relations or sad movies. Nelson [20] states that adult crying is mostly associated with grief, but makes a distinction between angry and sad crying. Also, lots of research has been done on crying behavior in infants[3], which is usually related to a need for attention, however in this report we will focus on crying in adults.

Equally important for this project is the research that is done on the phenomena of crying. This gives us grounds for including or excluding certain phenomena in our simulation, based on
their importance in crying. Tears are a very important aspect of crying. This is explained by Vingerhoets in [23]. According to him, people will more easily identify the emotions associated with crying if the subject produces tears. Another phenomenon of crying is sobbing. Sobbing is the convulsive inhaling and exhaling of air with spasms of the respiratory muscle groups. This is explained in [21]. Other phenomena include an increased heart rate and flow of blood to the head, as researched in [1]. Flow of blood to the head would result in a person blushing or their face getting more red. The results of this research are summarised in Table 2.1. In this table we rated the various aspects of crying on how important they are and on how difficult they are to implement. Importance is based on the visibility of the aspect and on human reaction on those aspects. For example, tears are very important because they trigger an emotional response so it is very important to implement, whereas increased heart rate is hardly perceived by an observer, so it is less important to implement. Also, we listed the emotions and other contributing factors for each phenomenon.

<table>
<thead>
<tr>
<th>Phenomenon</th>
<th>Primarily happens with</th>
<th>Importance</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facial Expression</td>
<td>Every emotion</td>
<td>Very</td>
<td>Easy</td>
</tr>
<tr>
<td>Tears</td>
<td>Sadness, anger</td>
<td>Very</td>
<td>Hard</td>
</tr>
<tr>
<td>Increased Heart rate</td>
<td>Stress, anger</td>
<td>Almost none</td>
<td>Hard</td>
</tr>
<tr>
<td>Blushing</td>
<td>Shame, anger</td>
<td>Average</td>
<td>Easy</td>
</tr>
<tr>
<td>Sobbing</td>
<td>Sadness</td>
<td>Average</td>
<td>Average</td>
</tr>
<tr>
<td>Red eyes</td>
<td>Tears</td>
<td>Average</td>
<td>Easy</td>
</tr>
</tbody>
</table>

Table 2.1: This table lists the different phenomena of crying, along with their contributing factors, importance and difficulty to implement.

Facial expression is very important and can be achieved by a facial animation engine. We will discuss earlier work that is related to facial animation in Section 2.3 and describe the technical details in Chapter 3. Sobbing can be seen as part of the facial animation. Tears are also important when simulation crying, and the related work will be discussed in Section 2.4. The details of our approach are described in Chapter 4. Techniques for simulating blushing and red eyes will be discussed in Chapter 5.

2.3 Facial Animation

As we have discussed in the previous section, emotions are an important part of conveying a realistic crying face. The most important visual effect of an emotion is the facial expression. To generate a facial expression we need at least a deformable model of a face. This can be either a model based on splines or be a traditional polygonal mesh. The latter is more conventional, although there are facial animation methods that are based on spline deformation [13]. Usually, computer games use a polygonal mesh to represent models. These models used to be animated by morph vertex animation (or per-vertex animation). This meant that an animation was stored in the mesh as a series of vertex positions, and was played frame by frame like a movie. These different positions were often created by attaching a skeleton to the mesh, and then creating the different key frames by putting the skeleton in different poses. This system was computationally very inexpensive, since it just had to play an animation frame by frame. These days, skeletal animation is more common to deform meshes in video games. A system of rigid bones is attached to the model and stored with the model. Each bone has a weighted influence on some vertices, and new positions of a vertex are calculated by a weighted blending of the
bones affecting a vertex. This is shown in Figure 2.1. The bones can be animated with skeletal animation files, describing the motion of each bone. This method is more flexible and memory efficient, but is computationally more expensive. There exist some methods that use spline-based skeletal animation, as opposed to a rigid skeletal system, such as [9, 8].

![Figure 2.1](image)

Figure 2.1: A simple model of an arm displaying bone animation. The bones have a weighted influence on the vertices. Note that the vertices of the elbow are affected by both bones.

Another method of animating a mesh is by using pose animation [5]. This actually resembles vertex animation in that it stores the movement of a vertex as a key frame in the mesh. But in pose animation a key frame, called a pose, describes the offset of one or more vertices to their original positions. When creating several poses, these poses can be blended because they are offsets of the same set of vertices. For example, a mesh can store a two poses of the same face, one with an angry expression and one with a sad expression. An example of pose blending can be seen in Figure 2.2. These poses can then be blended to show an expression of an emotion that is a mixture between sad and angry. This method is already used for facial animation [5, 7].

![Figure 2.2](image)

Figure 2.2: Pose Blend Animation. The face on the left is the neutral pose. The next two faces express anger and sadness. The face on the right is a mixture between anger and sadness. Notice the "angry" eyebrows and the "sad" mouth.

Pose blending allows us to easily generate diverse facial expressions. But we also need a way
to convert real-life expressions to a set of parameters suited for our simulation. Many times, for example in movies, this is done through motion capture. By attaching markers to an actors’ face, its movements are tracked and recorded. These can then be applied to a digital model of a head, either through some kind of bone animation or pose blending. This means there is a limited number of expressions available and an expression will always look the same, every time it is displayed. So we would like a way to exactly describe expressions, so we can control and generate them in a game interface for example. This way, we can create expressions without having to actually motion capture them ourselves, or use existing data and alter it if we want different expressions. A method to describe and generate facial expressions is the MPEG-4 standard [35] for facial animation. This substandard of MPEG-4, developed in 1999, is very efficient because it allows very flexible deformation of a face with a minimum of parameters. Like with body animation, it defines certain points on the face and describes the displacement of these points. This movement is described relative to a neutral pose, and every displacement of a point is given along a predetermined direction of movement. The combination of the direction and amount of displacement is called a FAP (facial animation parameter). The MPEG-4 Facial Animation standard has been used before for generating facial expressions in real-time. There is a method using facial animation tables (FATs) [10, 11]. This method creates tables which directly translate a FAP into the deformation of vertices on the face. This technique most resembles vertex or pose animation. Another way of using MPEG-4 Facial Animation is by using feature point mesh deformation [14, 15]. This directly links the FDPs to the vertices and transforms the mesh in real-time. This sort of animation is therefore comparable to skeletal animation.

Another system, called FACs [28], describes facial action units effective on the muscles of the human face. Examples of these actions are ”Lip Tightener” or ”Brow lowerer”. It was developed as for psychology purposes, to recognise and describe expressions and to link them to emotions. Later, it was adapted to be used in movies, to generate realistic expressions on computer generated images. It is more complex and less flexible than the MPEG-4 Facial Animation standard, although it may be more accurate. It requires the simulation of the face with its underlying muscles. Because of this complexity, it is an unsuitable candidate for real-time implementation.

The methods discussed in this section can be used for recreating expressions by applying geometrical deformations on a facial mesh. But if extreme emotions are to be simulated realistically, we need to look at the other aspects of these emotions we discussed in Section 2.2. These aspects are not incorporated in standards such as MPEG-4 Facial Animation or FACs, and can not be recreated by mesh deformations. To simulate tears we need techniques used in fluid simulation, which we will discuss in Section 2.4.

2.4 Tear simulation

As we have established in Section 2.2, tears are an essential part of crying when someone is sad or angry. We want the generated tears to be interactive with the user and the surroundings, as opposed to preprogrammed, to increase immersion. Therefore, using a texture will not be sufficient to generate tears in an interactive environment. So to simulate tears, we need to simulate fluids in a realistic way. In this section, we will discuss research on fluid simulation.
2.4.1 Fluid Simulation

Fluid simulation can be done in different ways, but the most commonly used approaches are grid-based (Eulerian) and particle-based. Grid-based simulations, such as [6], use a grid of points connected by springs. For each point at each time step, the next position is calculated using Euler integration. Grid based simulations for liquids usually involve Navier-Stokes equations to calculate the momentum of the fluid, and need additional formulas to conserve the mass and energy of the system. These computations can be quite complex. Grid based methods are more commonly used for creating large bodies of water or simulating water surfaces. The grid based approach is not very suitable for simulating drops of water, since multiple meshes are required. A more popular method for simulating such types of fluids is based on particles.

The Smoothed Particle Hydrodynamics (SPH) approach [17] is a particle-based method that uses particles with a fixed mass. Each particle represents a volume, which is calculated by dividing the mass by the density of the particle. SPH is a very general method which can be used for any application where field quantities have to be calculated. Müller [18] proposed a method where he applies SPH to fluid simulation. He uses SPH to solve a simplified version of the Navier-Stokes Equations, which describe the dynamics of fluids. By calculating the density of the fluid at discrete particle locations, the pressure and viscosity of the fluid, as defined by the Navier-Stokes Equations, can be evaluated anywhere in space. An additional advantage of the SPH method is that the simulation can be partially implemented on the GPU [12]. While the physical behavior is simulated by the particle system, the visualisation step is generally done by applying a point splatting technique [27] or a marching cubes algorithm [16], which we will discuss later in this section. SPH is a suitable approach for simulating the hydrodynamic behaviour of small bodies of water (like drops or tears) although it does not take interaction with a surface into account. Another challenge lies in integrating this method into a facial animation engine, in a way that allows an animator to control the fluid behavior as a part of the facial animation. Finally, real-time performance is a requirement for such an integrated system.

Shape matching [19] is a mesh deformation method developed for materials like rubber. It uses a particle system instead of a grid, but it uses a modified Euler scheme to apply the deformations. Furthermore, the particles are used so deform an existing mesh. It uses two sets of weighted particles, one representing the original shape, and one representing the changed shape. When the changed shape is changed, the original shape tries to match the changed shape, while trying to minimize the translation and rotations of its particles. A specific application of it, Fast Lattice Shape Matching [22] allows for the particle system to be split up into several parts (which we want when we try to simulate a fluid). With fast lattice shape matching, the mesh is given and is filled with voxels, used as particles, to control the deformations. However, this technique does not allow the mesh to be put together again.

2.4.2 Fluid Visualisation

As mentioned before, when using a particle based method, we need a way to visualise the fluid. The motions of the fluid are based on particles, and particles themselves have no connectivity information or normals. In some computer games, water splashes are sometimes imitated by representing the particles with sprites or simple geometry. These methods will not seem very convincing when applied to something like crying. For a realistic simulation we require more a more detailed representation of the surface of the fluid.
Several methods exist to recreate or visualise a mesh for a set of points or particles. Some of them, like [27], are only suited for identifying and recreating surfaces when a point cloud only contains points on the surface of the object. Fortunately, the particles on the surface can be identified. Other visualisation methods, like marching cubes [16] can be applied to render any point cloud or iso-surface. Müller suggested either point splatting or the marching cubes algorithm. Point splatting is the fastest and most efficient way of rendering the surface.

Point splatting [27] uses the position and normal of the points in a point cloud to visualise a surface. It projects the points on the screen space, just like with a normal polygonal mesh. For the other pixels it uses the coordinates and normals of projected points in the vicinity of the pixel to determine the color. This way, every point in the projected space becomes a function of points in its vicinity. In Figure 2.3 we show an overview of the point splatting technique. The disadvantage of point splatting is that it does not generate a mesh that can subsequently be rendered by a render engine. It requires a modification to the render engine to visualise the point cloud.

![Figure 2.3: An overview of the point splatting technique [27]. The visible points are projected on the screen space. For all other points on the screen space, kernels around the projected points are used to determine if that point is part of the object.](image)

Another technique, called the marching cubes algorithm, can be used to visualise the surface of a scalar field. Since SPH uses the particles to calculate scalar fields, this method can be used to visualise any scalar field generated by the particles. A scalar field which merely shows the influence of a particle would be sufficient to approximate the surface of the fluid. The algorithm traverses the scalar field through a grid. At each point on this grid, the algorithm samples 8 neighboring locations, forming an imaginary cube, then moves to the next ‘cube’ on the grid. Each of the points on a cube has a value in the scalar field. This value, when compared to a threshold value, determines if a point is inside or outside the surface. This information can be used to construct a small bit of the surface at each cube. By traversing the grid and creating the right polygons for each cube we can create a mesh which resembles the surface.

The marching cubes technique is mainly used for medical purposes, where it is used to visualise data from MRI or CT scans, as shown in Figure 2.4. In Figure 2.5 we show the difference between point splatting and the marching cubes algorithm.
Figure 2.4: An example of a head reconstructed from MRI scan data using the Marching Cubes algorithm. You can clearly see that the mesh is constructed by using cubes to determine the surface. Source: [39]

Figure 2.5: On the left is a glass of water rendered with the point splatting technique. On the right, the same glass of water is rendered with the marching cubes algorithm. Source: Müller [18].
2.5 Motivation and Goal

In this section we will motivate our research based on the techniques discussed in the previous sections. As shown in Section 2.3, there is plenty of research done on facial animation. Most of these methods only cover the animation of the face itself, by using motion captured emotions and/or a facial animation standard to describe the expressions. These methods do not handle any other effects when simulating extreme emotions, like crying. Less work is available on fluid simulation. Although there are some good techniques available for fluid simulation, the applications for these techniques in real-time 3D in particular are relatively new. Grid based techniques are mostly designed for simulating surfaces of water and not suitable for recreating tears. Particle based techniques are usually unsuited for recreating fluids in small quantities, such as tears, and tend to get unbalanced. The possibilities for gaming and simulation purposes are little explored. Furthermore, most games use preprogrammed emotions, which either use no crying at all, or a cartoonesque approach, like in the Sims for example. The tech demo of the game Heavy Rain, which has yet to be released, showed a girl crying in real-time 3D. These emotions were motion-captured, and the paths of the tears preprogrammed. Emotions that can be generated on the fly and respond to the user’s actions will result in a greater sense of immersion. Affective computer-controlled characters are very important in any story driven game or simulation.

Given these motivations, we will formulate the goal of this thesis project. Our goal is to create a real-time simulation in which a virtual character can display a wide range of emotions and generate tears on the fly. These emotions should not be preprogrammed but adaptable, since preprogrammed emotions decrease the realism of the simulation. The tears should also be generated as opposed to preprogrammed. Preprogrammed routines in virtual characters quickly become boring and reduce the realism and immersion of the game or simulation. To accomplish this goal we will look at facial expression and tears separately. We will handle the first problem, that of generating facial expressions, in the next chapter. We will discuss the generation of the tears in Chapter 4, where we will look into fluid simulation. We will also describe our own additions to the existing fluid simulation technique we used. The combination of facial animation and fluid simulation will be shown in Chapter 5, where we will discuss issues related to this integration. Chapter 6 presents the results of our simulation, where we will show various images of generated tears and look at the performance. Finally, in Chapter 7, we will draw our conclusions about the simulation and discuss possible future work that can be done for it.
Chapter 3

Facial Animation

3.1 Introduction

In the previous chapter we mentioned several important aspects of crying. One of these aspects is facial expression. For the generation of expressions we need a facial animation engine, which we will describe in this chapter. First, in Section 3.2, we will discuss a standard for describing facial expressions and how we plan to visualise these expressions. In Section 3.3, we present the technical details of the implementation and the problems we encountered with it.

3.2 Facial Animation

In this section, we will explain the MPEG-4 Facial Animation standard. We favor the MPEG-4 Facial Animation standard to describe the facial expressions in our crying simulation. The reason behind this is because the standard is well documented [35] and there are several papers that have implemented a facial animation engine based on MPEG-4 FA [10, 11, 14, 15]. Also, there are a lot of other applications that use the MPEG-4 FA standard [33, 2]. The animation data of these applications can also be used in our application for future research. Furthermore, the alternative method, FACs, demands a too complex facial model and control structure to be easily implemented for real-time purposes. Another option would be using motion captured expressions, which do not necessarily comply to a specific standard. Because we aim to propose a technique that supports user interaction, we need a more flexible solution. For example, if a user has effect on the virtual character’s mood, we want their facial expressions to react accordingly. If there are only a few preprogrammed expressions, the user will soon lose affection with the virtual character. If the expression can be generated from a set of parameters that define mood of a character, the expressions will be more diverse and appropriate for the situation. Now that we have explained why we chose MPEG-FA we will continue to describe this standard.

The MPEG-4 Facial Animation standard is based on key-points that define the shape of a persons face. The standard defines 84 feature points called Facial Definition Parameters (FDPs). These points are necessary for the standard to identify which facial features are located where. For example, FDP 9.3 defines where the tip of the nose is. These FDPs are subdivided into regions they belong to, for example, the right eye or the mouth. The FDPs are shown in Figure 3.1. The black ones are used for animation purposes, the white ones are only used for calibration of the face.
Typically, these feature points are defined by hand once, in the neutral pose of the face. The neutral face is defined by the MPEG-4 FA standard as follows:

- the coordinate system is right handed; head axes are parallel to the world axes.
- gaze is in direction of Z axis.
- eyelids are tangent to the iris.
- the pupil diameter is one third of the iris diameter.
- lips are in contact; the line of the inner lips is horizontal and at the same height of lip corners.
- the tongue is flat, horizontal, with the tip of the tongue touching the boundary between upper and lower teeth.
Our neutral face is shown on the left in Figure 3.5. For animation, the standard defines a set of Facial Animation Parameters. These parameters represent the displacement of sets of FDPs. They are divided in low- and high-level FAPs. High-level FAPs are used to represent, with a single parameter, the most common facial expressions (joy, sadness, anger, fear, disgust, surprise) and mouth postures. Low-level FAPs control facial features individually. Each low-level FAP affects one FDP, except for the FAP that is responsible for rolling the tongue, which affects two FDPs. What FAP controls what FDP is defined in a table. Also included in this table is the direction in which the FDPs are moved. As an example, part of this table is shown in Table A.1. Note that some FAPs are bidirectional, so they have a negative and positive maximum in our implementation.

<table>
<thead>
<tr>
<th>nr.</th>
<th>FAP name</th>
<th>FAP description</th>
<th>Units</th>
<th>Uni / Bidir</th>
<th>Motion</th>
<th>Grp</th>
<th>Subgrp</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>lower_t_midlip</td>
<td>Vertical top middle inner lip displacement</td>
<td>MNS</td>
<td>B</td>
<td>down</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>raise_b_midlip</td>
<td>Vertical bottom middle inner lip displacement</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>stretch_l_cornerlip</td>
<td>Horizontal displacement of left inner lip corner</td>
<td>MW</td>
<td>B</td>
<td>left</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.1: Part of the table that contains the FAP definitions.

The amount that an FDP is displaced by a FAP is measured in Facial Animation Parameter Units (FAPUs). There are 6 FAPUs. Each FAPU is based on fractions of key facial distances (e.g. the distance between the eyes). The FAPU are listed in Table 3.2. The distances between the facial features are shown in Figure 3.2. The standard defines which FAP uses which FAPU, as seen in Table A.1. For example, FAP number 4 moves the FDP top middle inner lip FDP and uses MNS as a unit for moving this FDP. This FAPU is defined as the mouth-nose separation, and is based on the distance between the mouth and the nose, called MNS0 in Figure 3.2.

<table>
<thead>
<tr>
<th>Description</th>
<th>FAPU value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRIS Diameter in neutral face</td>
<td>IRISD = IRISD0 / 1024</td>
</tr>
<tr>
<td>Eye Separation</td>
<td>ES = ES0 / 1024</td>
</tr>
<tr>
<td>Eye - Nose Separation</td>
<td>ENS = ENS0 / 1024</td>
</tr>
<tr>
<td>Mouth - Nose Separation</td>
<td>MNS = MNS0 / 1024</td>
</tr>
<tr>
<td>Mouth - Width Separation</td>
<td>MW = MW0 / 1024</td>
</tr>
<tr>
<td>Angular Unit</td>
<td>AU = 10-5 rad</td>
</tr>
</tbody>
</table>

Table 3.2: The table listing the five different FAPUs

We need to translate the FAP values, which describe a facial pose or expression, to a mesh representation of a human face. We do this by assigning the FDPs to our facial model. For every feature point (FDP) in the model we compute its neighboring vertices, and the surface distance to those feature points. With this information, we compute weights for each vertex in the model. A smaller distance between a feature point and a vertex results in a higher influence of that feature point on the vertex. A vertex can be influenced by multiple feature points. The
We propose to use pose vertex animation for our implementation of the MPEG-4 Facial Animation standard. We made this decision because it is computationally inexpensive, which makes it very suitable for real-time 3D facial animation. Also, it is already successfully implemented for facial animation purposes [5, 7] and it is supported by most graphic engines. In short, it allows blending between different kind of "poses" which are a series of offsets to the base vertex data. When we move a vertex in a modelling program, we can calculate its offset to its original position and create a pose. We use control points for the FDPs in our modelling program and attach vertices to it with a certain weight. We move an FDP, that FDP influences vertices and we translate the movement of an FDP into a pose. Since the movement of an FDP is determined by a FAP, we can translate FAPs into poses, as shown in Figure 3.4. By using control points in the modelling program we preserve the behavior of an FDP and by exporting their movements to poses we avoid having to deal with control points or bones in our final implementation.

We can use FAPs to define several expressions which are useful to our research (expressions that are related to crying). We convert these expressions from FAPs to poses, so we can display them using a mesh, and blend them. This is similar to the example shown in Section 2.3, Figure 2.2. However, we chose to model 68 separate poses, one for each FAP. For each FAP, we move its corresponding FDP from its original position in the neutral face in the direction defined by the MPEG-4 FA standard. We move it one unit of its corresponding FAPU. The
The movement of the FDP causes a movement of the nearby vertices, which are influenced by the FDP, as previously explained. We store the offset of the moved vertices as a pose in the mesh. This gives us 68 poses with which we can make every expression by blending the poses. An example of this is shown in Figure 3.5, where we move FAP number 5 and store the result as a pose in the original model. These 68 FAPs together describe an expression. A static expression alone is not enough to display a convincing facial animation. We can interpolate over time between different expressions, to create an animation. Animation is an important aspect of displaying extreme emotions. As shown in Section 2.2, an aspect of crying is sobbing, the convulsive inhaling and exhaling of air. Displaying this is achieved by animating the face accordingly.

![Diagram of the conversion from a FAP to a mesh representation in the graphics engine.](image)

**Figure 3.4:** The conversion from a FAP to a mesh representation in the graphics engine.

*Figure 3.5:* On the left is our neutral face. On the right is our neutral face with FAP 5 set to its (negative) maximum value.
3.3 Implementation

We used Ogre [36] for visualising our method. Ogre is a widely used open source graphics engine, capable of vertex pose animation. We modelled the head in Maya, which can export a .mesh file, the format that OGRE uses. After we modelled the head, we created a set of bones, resembling the FDPs. These bones were then attached to the mesh by weighing each bone for each vertex. The influence of a bone was determined by a quadratic function of the distance between the bone and the vertex up to a maximum distance. This distance is measured over the mesh, and not euclidean. We set this distance so that most vertices were affected by two or three bones. We did not do this by hand, but Maya can do this automatically. After this procedure, we had to tweak a lot of weights, because the FDPs of the mouth were really close together. These FDPs were affecting too much vertices in the whole mouth region, causing weird and uncontrollable motions. The reason we did most of the weighing by hand is because there is no flawless way of doing it completely automatic, although a few techniques have been suggested in the MPEG-4 FA based methods we mentioned earlier [10, 11, 14]. Most professional animators produce far more successful results doing it by hand and actually prefer this, giving them more control over their work. Also, since attaching the FDPs is done in the modelling process, there is no need to do it quick. When the bones are attached, the different poses can be created. Because we make one pose for each FAP and each FAP motion is documented, there was no need doing this by hand. The way Maya creates poses for a model is by using another model as reference. So we made a script that did the following:

**Algorithm 1 Create Poses**

Require: A facial mesh rigged with FDPs

for i = 1 to 68 do
   Create a copy of the mesh and bones
   On the copy, move the FDP indicated by FAP i in the defined direction and distance.
   Create a pose on the original model, using the copy as a reference
   Delete the copy

This gave us the original mesh with all the FAPs as poses. We could then export this as a mesh, and load in to OGRE. To generate FAPs we built an application. We have built an application with 68 sliders, one for each FAP. Some of these, as defined by the standard, are bidirectional, meaning that the value of the slider can be negative or positive. Each slider corresponds to a pose, which represents a FAP, like we explained in the previous section. Because the pose animation allows blending of several poses, we can create any expression we want by combining different poses, exactly how we would describe the expression by using FAPs. When a desired expression is created, it can be stored in a FAP file. When storing one expression, the user has the option of associating a frame or time with this pose. When different poses are stored with different time labels, the application can then linear interpolate between these poses and create an animation. It generates 22 frames per second, which is enough for a smooth motion, and stores them in a FAP animation file. A screenshot of the application can be seen in Figure 3.6. These files can then be read by another program, which just loads FAP animation files, and displays the different poses frame by frame. Figure 3.7 shows different facial expressions created with the editing program.
Figure 3.6: A screenshot of our application where we can create FAP based animations.

Figure 3.7: Various expressions created with the FAP editor.
3.4 Summary

One of the aspects of displaying extreme emotions is displaying expressions, as discussed in Section 2.2. We have shown several methods and two standards for facial animation in Section 2.3. In this chapter we have discussed the technical details of the MPEG-4 Facial Animation as well as the implementation of this standard. This provides us with a way of animating the expressions of a human face. Facial animation is just one of the aspects needed for displaying extreme emotions. In Chapter 4 we will discuss a method to simulate tears, which is needed for a convincing crying simulation, as shown in Section 2.2.
Chapter 4

Fluid Simulation

4.1 Introduction

In the previous chapter, we discussed the details of our facial animation engine. As mentioned in Section 2.2, facial animation alone is not enough to fully simulate the extreme emotion of someone who is crying. Another important aspect is the production of tears. To recreate tears in a realistic and flexible way, we want to simulate a fluid that can interact with the face in real-time. In Chapter 2 we have described various techniques to achieve this. We will discuss Smoothed Particle Hydrodynamics (SPH) and the method proposed by Müller et al.[18] in the next section. In Section 4.3 we will describe how we visualise the fluid. We have created our own addition to the fluid simulation, fluid trail synthesis, which we will also discuss in Section 4.3. Finally, we will discuss the implementation details in Section 4.4. A schematic overview of the fluid simulation can be seen in Figure 4.1

Figure 4.1: A schematic overview of our fluid simulation engine. The parts marked in blue include our additions to the existing method.

4.2 Smoothed Particle Hydrodynamics

In this section, we will first briefly explain the SPH-based method proposed by Müller et al.[18]. The SPH method consists of a few basic steps to be executed for each frame. First, the densities of the particles need to be calculated based on their current position. Then,
the pressure and viscosity forces of the Navier-Stokes equations have to be calculated for each particle. In [18], also an additional surface tension force is suggested for better fluid stability. Finally the external forces, such as gravity and collision have to be applied to the particles. Originally only gravity and collision are applied, but for tear simulation we have found two other forces necessary for a stable simulation. First, we apply friction to the particles, so the tear will stop rolling after a while. Second, we added an adhesion force, so the fluid will stick to the surface. We will discuss these additions at the end of this section. Then, a scalar field based on the location of the particles is defined to estimate the surface and the normal of the simulated fluid. Following [18], we will call this scalar field the color field in the remainder of this report. The color field can be visualised using a mesh generation algorithm, such as marching cubes [16] or point splatting [27]. This will be discussed in Section 4.3. First, we shall explain the hydrodynamic forces of the fluid simulation in more detail. A schematic overview of the steps involved in our SPH-based method can be seen in Figure 4.2.

Figure 4.2: A schematic overview of our modified SPH-based method. These steps are repeated each frame of the simulation.

The method described by Müller et al.[18] is based on Smoothed Particle Hydrodynamics (SPH). Smoothed Particle Hydrodynamics is a method that was originally developed as a simulation for astrophysical problems. The hydrodynamics of a fluid can be seen as velocities that need to be evaluated anywhere in space. This is done using the Navier-Stokes equations. We will not go into detail on the Navier-Stokes equations here, but it evaluates velocities of the fluid at any location based on three terms. The first term calculates the movement of the fluid caused by the change of pressure. The second term calculates the effect of viscosity, which is the second derivative of the velocity of the fluid multiplied by a viscosity constant. The third part consists of the external forces effective on the fluid. The velocities calculated by the Navier-Stokes equations can be seen as a vector field. In mathematics and physics, a vector field associates a vector to every point in a space. This is similar to a scalar field, which associates a scalar value, also called a field quantity, to every point in a space. For example, the distribution of pressure in a fluid can be seen as a scalar field. Because these are continuous functions, they need to be discretised.

Smoothed Particle Hydrodynamics uses discrete particle locations at which field quantities are defined. The value of the field can then be calculated anywhere in space using smoothing
kernels, which distribute the quantities in the neighborhood of a particle. A smoothing kernel is a function which returns a value for a position anywhere in space based on the distance of that position to the center of the smoothing kernel. A scalar quantity $A$ at location $\mathbf{r}$ is defined by a weighted sum of all the particles as shown in the following formula:

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h).$$  \hspace{1cm} (4.1)

where $j$ iterates over all particles, $m_j$ and $\rho_j$ are the mass and density of particle $j$ and $\mathbf{r}_j$ its location. The scalar quantity $A_j$ is the discrete field quantity at location $\mathbf{r}_j$. The function $W(\mathbf{r} - \mathbf{r}_j, h)$ is called the smoothing kernel with a maximum influence radius of $h$. The smoothing kernel gives us a value based on $r$, the distance between $\mathbf{r}$ and $\mathbf{r}_j$. This means that the smoothing kernel are centered at the particle $j$. Any $r$ larger than $h$ will produce a value of 0, meaning that particle $j$ has no influence on location $\mathbf{r}$ if the distance between them is more than $h$. The gradient of a smoothing kernel, $\nabla W(\mathbf{r}, h)$, gives us a vector in the direction of the center of the kernel. This gradient can be seen as the change in influence of this particle, and the direction of this change. This is needed in some equations where derivatives of a field quantity are needed. To find the density at any location $\mathbf{r}$ in a scalar field, the following function is used:

$$\rho(\mathbf{r}) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h).$$ \hspace{1cm} (4.2)

We use the location of a particle for $\mathbf{r}$ to get the density of a particle. So before calculating the forces, we first need to calculate the density of all the particles.

According to [18], particles have three forces acting on them: the pressure, the viscosity and finally the external forces. The main external forces used in our simulation are gravity and friction. The pressure force on a particle is the derivative of the pressure at the location of the particle. By applying Equation 4.1 and using $p$ for the scalar quantity, the pressure force on particle $i$ is calculated as follows:

$$\mathbf{f}_{\text{pressure}}^i = - \sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h).$$ \hspace{1cm} (4.3)

The pressure values at the particle locations, $p_i$ and $p_j$, are calculated by multiplying the density $\rho$ of the particle with a gas constant $k$. Note that we use the derivative of the smoothing kernel, since this results in the direction of the change of pressure. Also, [18] made some modifications to this formula to achieve a more stable simulation. For the viscosity force equations, we also need to know the velocity of the other particles. The viscosity causes particles to adjust their velocity to match each other, and is present in the Navier-Stokes equations with the term $\mu \nabla^2 \mathbf{v}$, where $\mu$ is the viscosity constant for the fluid. Using SPH, the viscosity force can be found using the following equation:

$$\mathbf{f}_{\text{viscosity}}^i = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h).$$ \hspace{1cm} (4.4)

where $\nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h)$ is the derivative of the gradient, or: the Laplacian of the kernel. The velocities of the particles $i$ and $j$ are given by $\mathbf{v}_i$ and $\mathbf{v}_j$. The direction and effect of the pressure force and forces are best seen in Figure 4.3.
Müller also defines another force called the surface tension. This force is not in the original Navier-Stokes equations. It is used to balance the forces on the surface of the fluid, so that a cohesive fluid is guaranteed. Normally in a fluid, all the molecules are pulled toward each other but because the forces come from every direction they balance each other. The molecules on the surface, on the other hand, are only pulled by the particles ‘below’ them, causing them to be pulled inwards. This is the surface tension of a fluid. In order to calculate the surface tension force, a representation of the surface represented by the particles is needed. This surface is estimated by defining the color field, which is 1 at particle locations and 0 at every other location. By applying Equation 4.1 and substituting $A_j$ by 1, a smoothed color field can be computed at any location with the following equation:

$$c_s(r) = \sum_j m_j \frac{1}{\rho_j} W(r - r_j, h).$$  \hspace{1cm} (4.5)

The gradient of the color field can be computed at any location ($r$) in the color field, but the gradient will only be significant near the surface of the fluid. Inside the fluid, because there are a lot of particles, the color field will be 1 or almost 1 everywhere, so the gradient will be small. Near the surface, the color field will go from 1 to 0 over the course of $h$, the smoothing kernel size. So the color field gradient $n$ of the smoothed color field, shown in Equation 4.6, will point inward in the direction of the normal of the surface. The surface normals can be determined by evaluating $n$ near the surface, normalising $n$ and inverting it. The smoothed color field can later also be used for identifying and visualising the surface of the fluid.

$$n = \nabla c_s.$$  \hspace{1cm} (4.6)

The surface tension depends on the curvature of the surface $\kappa$ and a tension coefficient $\sigma$, which depends on the two materials (in this case air and water) that form the substance. The curvature of the surface $\kappa$ is the divergence of the surface normal, also the Laplacian of the smoothed color field, and is given by the following equation:

$$\kappa = \nabla \frac{-n}{|n|} = \frac{-\nabla^2 c_s}{|n|}$$  \hspace{1cm} (4.7)
And this leads to the formula for the surface tension field as follows:

\[ f_{i}^{\text{surface}} = \sigma \kappa \mathbf{n} = -\sigma \nabla^2 c_{s} \frac{\mathbf{n}}{|\mathbf{n}|} \]  

(4.8)

To calculate the surface tension on particle \( i \), we need to evaluate the surface normal and the Laplacian of \( c_{s} \) at location \( \mathbf{r}_i \), which is the location of particle \( i \). Note that this force should only be applied on particles near the surface, since the normals at other locations are meaningless.

Once the hydrodynamic forces for the particles are computed, we can apply the external forces. In the original SPH approach [18], only gravity and a collision force are defined. The method does not take friction into account. In the original method, after contact with a surface, a particle is simply sent in a direction mirroring its incoming direction on the surface. The reflected particle is kept in place by the particles above it, resulting in a stable simulation. A problem with this approach is that it works best with a large number of particles, which puts a strain on the real-time requirement of the fluid simulation. When dealing with a limited number of particles, in the case of tears, such an approach leads to an unstable fluid simulation. To solve this we do not reflect the particles after contact with a surface, but chose to apply friction to the particles as an additional external force in our simulation. A benefit of this is that the friction can be handled the physics engine.

Another external force that is missing is adhesion. In real life, the small molecular forces between the water and a surface cause the water to stick to the surface. The lack of skin adhesion causes the particles to lose contact with the surface as soon as the surface starts facing downward. An example of this is shown in Figure 4.4.

![Figure 4.4: A schematic side view of a face. The red line shows the trajectory of the particle. On the left is shown how it will probably go, without adhesion. On the right is the approximation of a trajectory with adhesion.](image)

We approximate this behavior by adding a force to the particles that are in contact with the skin surface. The direction of this force is in the opposite direction of the normal of the skin surface (thereby attracting the particles into the skin). The formula for this force is as follows:

\[ f_{i}^{\text{adhesion}} = -\alpha \frac{\mathbf{n}_s}{\rho_i} \]  

(4.9)

where \( \mathbf{n}_s \) is the normal of the skin surface and \( \alpha \) is a coefficient which determines how adhesive the surface is. Because the density \( \rho \) of a particle is taken into account, the attraction to the surface depends on the volume a particle represents. A lower density means a particle represents a larger volume, which means that a larger area of fluid contacts the skin, causing a stronger adhesion to the skin for the particle.
4.3 Visualisation

In the previous section we discussed how the physical behavior of tears are simulated using the SPH method, extended with friction and a skin adhesion force. The next step is to visualise the fluid by generating a mesh. There are two parts to visualising the fluid. First, we need to generate a mesh for the tear itself. Second, we need to simulate and visualise the trail a tear makes. The second part is not included in the method proposed by Müller and is our addition to small scale fluid simulation. A schematic overview of the steps of our fluid visualisation can be seen in Figure 4.5.

![Figure 4.5: A schematic overview of the fluid visualisation. The parts marked in blue are our addition to fluid simulation.](image)

4.3.1 Fluid Visualisation

In Section 2.4 we discussed two established methods to render the fluid. These are point splatting and the marching cubes algorithm. Point splatting seems the fastest and most efficient way of rendering the surface. We can calculate locations of particles on the surface, and we can calculate the normal of that surface by using the previously mentioned color field. This is all the information we need for point splatting to project the shape of the fluid on the screen space. We will explain how we determine what particles are on the surface using the color field.

As we mentioned before, the color field is a scalar field and the color field gradient $n$ can be calculated at any point with the following formula:

$$n = \nabla c_s.$$  

(4.10)

Only at the location of the particles on the surface will there be a significant color field gradient. Everywhere inside the fluid, the value of smoothed color field will be 1 or only slightly lower, so the gradient is insignificant. The gradient of the smoothed color field at particle locations at the surface is in opposite direction of the normal of the surface. This way, when the size of the gradient of the smoothed color field at position $r$ gets above a certain threshold we know that position $r$ is near the surface of the fluid, and not in the middle. When $n$ is normalised and inverted, when calculated at the location of particles near the surface, this gives us the normal for the surface. This means we can calculate all the information we need for point splatting. Unfortunately, point splatting requires us to modify the render engine we were using we decided not to use this technique and to use the marching cubes algorithm[16].

Like we mentioned in Section 2.4, the marching cubes algorithm can be used to visualise an isosurface of a scalar field, such as the color field generated by SPH. An isosurface is a surface
that represents points of a constant value, the iso-value, within our scalar field. The algorithm traverses the scalar field through a grid. At each point on this grid, the algorithm samples 8 neighboring locations, forming an imaginary cube, then moves to the next ‘cube’ on the grid. As an example, Figure 4.6 shows a grid with a cube expressed in the coordinates of the grid.

Figure 4.6: A grid with an imaginary cube progressing through it. The coordinates of the cube are expressed in i, j and k.

Each of the points on a cube has a value in the scalar field. As mentioned before, the isosurface is determined by the iso-value. If the value of a point is higher than the iso-value it is inside the surface, otherwise it is outside the surface. So a cube can be fully inside the surface, if all the points have a higher value than the iso-value, fully outside it, or partially inside it. Through rotations and reflections there are only 15 unique polygon configurations for a cube, as shown in Figure 4.7, formed by the points determined by the iso-values.

Figure 4.7: Possible configurations of a cube. [39]

By traversing the grid and creating the right polygons for each cube in a scalar field we can create a mesh which resembles the isosurface. In our case, we want to construct the isosurface for the smoothed color field. So when using the marching cubes algorithm, we need to calculate the smoothed color field value, $c_s$, at every corner of a cube while it progresses through the grid. A more precise position of the vertices on the edges of a cube can be calculated by linear interpolation of the scalar field values at the two corners of that edge. This is done with the following formula:

$$ p = p_1 + \mu (p_2 - p_1), \text{where } \mu = \frac{i - c_s(p_1)}{c_s(p_2) - c_s(p_1)}. $$

(4.11)
The new vertex to be created is \( p \), and the two corners of the edge are \( p_1 \) and \( p_2 \). The color field values at those locations are given by \( c_s(p_1) \) and \( c_s(p_2) \), and \( i \) is the iso-value for the surface. This formula calculates the position on the edge with scalar field value \( i \) by looking at the difference in color field values of the two corners of the edge and their distance. A two dimensional example of our smoothed color field and a marching cube can be seen in Figure 4.8.

![Figure 4.8](image)

Figure 4.8: A two dimensional example of the marching cubes algorithm (technically called a marching squares algorithm) applied to the color field. The values in the corners of the cube show the smoothed color field values at the corners of the cube. The iso-value for the iso surface in this example is 0.2. The red line shows the resulting surface. The blue areas show the influence of the smoothing kernels centered at the surface particles \( p_1 \) and \( p_2 \).

The next step is creating the right normals for each polygon. Since the mesh is created for the isosurface of the smoothed color field \( c_s \), we can use the gradient of the smoothed color field at the location of the vertices as the normal. This normal is calculated the same way as in the point splatting technique, by looking at the gradient of the color field. Using the visualisation technique described in this section, we can render the fluid in real-time. The motions of this fluid are calculated with the method explained in Section 4.2. An example of water calculated and visualised with the previously described techniques can be seen in Figure 4.9.

![Figure 4.9](image)

Figure 4.9: A small amount of water falling on a sphere. The water is visualised using the marching cubes algorithm.
4.3.2 Fluid Trail Visualisation

We have discussed the physical aspect of simulating fluids and a method of visualising the fluid. Unfortunately, the Smoothed Particle Hydrodynamics method that we use is not sufficient for the level of detail we need when recreating a tear. In real life, a water drop leaves a trail as it slides down a surface. An example of this is shown in Figure 4.10

![Figure 4.10: A person who is crying. It clearly shows the trails that are created by the teardrops that roll down his face.](image)

This is partly because of the adhesive force. Water molecules stick to the surface, decreasing the volume of the drop. A simulation with that level of detail requires a very large number of particles, so creating a trail by using particles is nearly impossible. Therefore, we created a method that simulates the trail and the decreasing volume of a tear. The method identifies all the different drops of the tear and then it stores their boundaries. These boundaries are then used to create a mesh which represents the trail.

Every step, we first identify the different tear drops and which particles are part of which drop. To do this, we use the same data structure we also use for calculating the SPH forces and the marching cubes algorithm. We will discuss this data structure in Section 4.4. To identify a drop, we take a random node from a list of nodes with particles in them. We add the particles in this node to the collection of particles that create this drop and flag this node as checked. Then we look at its neighboring cells and do the same if they contain particles. We repeat this until we do not find any new neighboring cells with particles in them, meaning we have found a complete drop. From the collection of particles that are part of this drop, we store the rightmost and leftmost particle. Then, we select a new random node from the nodes we have not checked yet and repeat the algorithm to find a new drop. We repeat this until we have run out of nodes.

For each drop, we now have the leftmost and rightmost position. We will call those the boundaries of the teardrop. We will use those points to create a mesh which represents the trail of the drop. First, we create a point in the middle of the boundary and we elevate this point slightly in the direction of the normal of the skin. The middle point can not be raised more than $h$, the smoothing kernel size. This would cause the trail to become thicker than the teardrop itself. The positioning of these points is best explained in Figure 4.11. The steps of the method are shown in Algorithm 2 and Algorithm 3. The result of this technique can be seen in Chapter 6.
Figure 4.11: A schematic top view of a teardrop. The particles which are marked red are the left- and right most particles of the drop. These are the basis for the left- and rightmost points of the trail. As seen, the middle point is slightly elevated, to make the trail appear to have a volume and lying on top of the skin.

**Algorithm 2** Find drops

**Require:** A set of nodes $N$ with particles in them

Let $P$ be an empty collection of particles
Select a random unchecked node $\nu$ from $N$
Find boundaries($\nu, P$)
Let $p_{\text{max}}$ be $\max(\forall p \in P)$ and $p_{\text{min}}$ be $\min(\forall p \in P)$
Find drops($N$)

**Algorithm 3** Find boundaries

**Require:** A node $\nu$, a collection of particles $P$

Add the particles in node $\nu$ to $P$
Flag $\nu$ as checked so that we do not check it again.
find neighboring nodes, $\nu'_1 \ldots, \nu'_n$ of $\nu$
for $i = 0$ to $n$ do
  Find boundaries($\nu'_i, P$)

Every frame, we create these three points. With those three points and the points from the previous frame, we create four triangles, which simulate a small part of the trail. If we do this every frame, a complete trail of the teardrop will be created. The progression of this method is shown in Figure 4.12.

When a drop splits into two new drops, both new drops use the boundary of the old drop, causing the trail to branch in two directions. Calculating a new part of the trail every frame will be a waste of triangles and computation time. So first, at every frame, we compute the new boundaries of the tear. We store the boundaries of this tear only if the distance $d$ between the current and previously stored center is larger than $d_{\text{min}}$, a variable we set beforehand. This variable resembles the minimum distance a teardrop has to travel before creating a new part of the trail. Only then do we create a new set of triangles for the trail, using the previous and the new boundaries of the drop.

A result of a drop leaving a wet trail is that it reduces in size because of the water molecules that stick to the surface. Because we do not simulate the trail with particles, we have no accurate way of diminishing the size of the drop. Because of this we alter the mass of the particles in the drop, based on how far the drop has traveled over the surface. Decreasing the
mass has two important effects. First, the mass has an effect on the density and pressure of the particle. This causes a particle to represent a smaller volume of liquid, making the teardrop smaller. Second, a lower mass causes friction to have a greater effect on the particle. So the further a tear travels the slower it will most likely go, depending on obstacles and gravity.

4.4 Implementation

Now that we have discussed all the technical details of calculating the motions of a fluid and visualising it, we will discuss the implementation of the several aspects involved in our simulation. In Section 4.4.1, we will first explain what applications we used for implementing our SPH based method and why. In Section 4.4.2 and Section 4.4.3 we will discuss the modified octree data structure we created for our implementation and discuss its effect on both the SPH method and the Marching Cubes algorithm.

4.4.1 Smoothed Particle Hydrodynamics

Since we used OGRE for our facial animation, we need a way to implement particles in this engine. OGRE does have a particle system, but it uses particle generators to generate particles that have a certain lifespan. The particle system used in SPH uses a constant number of particles which do not “die”. Such a system is not very hard to implement, since a particle in this system only has a position and a weight. We used OgreODE [37, 38], a physics engine for OGRE, to apply forces and collisions to the particles. We chose OgreODE because it offers an easy to use physics engine based on ODE and was created to be used for OGRE. Using an already available physics engine meant that we did not have to worry about implementing collisions, gravity and force mechanics ourselves, which can be quite complex and is not the
focus of this thesis project.

OgreODE uses movable and static bodies to represent physical objects. These bodies have a geometry and can be checked against each other for collision. In case of a collision, we can set the friction and bounciness of the collision. OgreODE also allows us to apply forces to bodies, causing them to move. This is very useful for the implementation of our SPH-based method. We used cubes to represent the particles bodies in OgreODE. We chose cubes because cubes do not roll across a surface they touch, creating torque, which further influences the movement of the particle. We use the position of these bodies for the location of the particles. This means that after calculating the forces, we apply them to the bodies, and the physics engine moves the bodies. When we need to know the new location of a particle, we check the location of the body that is assigned to the particle.

4.4.2 Data structure

In the previous sections we explained all the calculations needed for the simulation. Often, these calculations would require all the particles to be checked against all other particles. When checking all the particles against each other the simulation would become very slow, since the forces would be calculated in $O(n^2)$ time. This is also true for the marching cubes algorithm, where we need to calculate the smoothed color field, which is calculated by checking every particle. Fortunately, the use of smoothing kernels means that a particle will only have an effect on other particles in its neighbourhood, determined by the smoothing kernel size $h$. Therefore we put the particles in a data structure to efficiently look up the particles within range of the smoothing kernel. First, we used a uniform grid because that would also be useful when implementing the marching cubes algorithm later, for visualising the mesh. But since the water is moving and we do not know where it will be or how much space it will occupy, a grid can be very inefficient. To account for all the places the water could be in the simulation requires a large grid, making the number of empty cells very large. In our earlier fluid simulation tests, this worked very slow, so we decided to use an octree data structure. With this structure, every cell has a maximum capacity of how many particles it can contain. If the maximum capacity is reached, it divides the cell into eight equal size new child cells. We can also set a maximum depth to the tree. Since the root cell is a certain size, and all its children are half that size, we can tell what size the cells at the maximum depth of the tree are going to be. Having cells that are smaller than our smoothing kernel is not very useful, so we set the maximum depth to correspond with the smoothing kernel size. This means that cells with particles in them can still be larger than the smoothing kernel size, if the maximum capacity is reached before the maximum depth is reached. To prevent this, we set the maximum capacity at one, meaning a cell will not stop splitting until the maximum depth is reached. All the cells with particles in them will be the same size as the smoothing kernel, so we only have to check the particles in the neighboring cells when calculating the forces on a particle. It turned out that finding the neighboring cells was still too slow and caused other problems with the marching cubes algorithm, which we will discuss later. For example, finding the neighboring cells can be still be a relatively expensive operation if a neighboring cell happens to be on the other side of the tree, like in Figure 4.13.

To solve this problem we slightly change the octree data structure. We add a particle to a node if it is within distance $h$ of a node, where $h$ is the size of the smoothing kernel of a particle. This allows particles to be assigned to several nodes, but each particle is within the real boundaries of only one node. This is shown in Figure 4.14.
Figure 4.13: On the left we see part of a quadtree (a two dimensional octree), with several particles in two nodes. Looking up a neighboring position of the green node still requires us to traverse all the way down, in case of the red node. This is shown on the right.

Figure 4.14: This tree shows our modified octree data structure. Empty node $\nu$ now includes particles that are within $h$ of the boundaries of a node, where $h$ is the smoothing kernel size of a particle. In this example, $h$ is half the size of a node, but both the node size and $h$ are variable.

This way, when we check the particles of a certain node, we only need check them against the other particles that belong to the node. To avoid checking the same particle several times (since a particle can belong to several nodes now), we keep track of which particles of a node are actually inside the node. For each particle, we also keep track of its parent node, which is the one node it is actually inside of. Since we no longer have to look for the neighboring nodes, we made several calculations much faster, for both the marching cubes algorithm and the force calculations.

We will show the algorithm used when maintaining the octree, before and after our modifications. The algorithm is used when a new particle is created or when an existing particle moves into a new node. To assign a particle to a node we first used the following algorithm:
Algorithm 4 AssignParticle

Require: A particle $p$ and an octree node $\nu$

if particle $p$ is not in node $\nu$ then
    return

if depth of node $\nu$ is smaller than maximum depth $d$ then
    Let $\nu'_1, \ldots, \nu'_8$ be the children of node $\nu$
    for $i = 1$ to $8$ do
        AssignParticle($\nu'_i$, $p$)
    else
        assign $p$ to $\nu$

This algorithm is quite straightforward. It looks at a particle and checks if it is inside a node, usually starting with the root node. If it is inside the node, it will check if it is in one of the children of that node, moving recursively through the tree. It stops when it has reached a node at the maximum depth of the tree, and then assigns the particle to that node. This way, every particle would belong to only one node, and that node would be at maximum depth $d$, and at depth $d$, we know the size of the nodes. But because nodes now contain all particles within range $h$, we have to slightly change the algorithm:

Algorithm 5 AssignParticleModified

Require: A particle $p$ and an octree node $\nu$

Let $A$ be the area of node $\nu$ expanded by length $h$

if particle $p$ is not in $A$ then
    return

if depth of node $\nu$ is smaller than maximum depth $d$ then
    Let $\nu'_1, \ldots, \nu'_8$ be the children of node $\nu$
    for $i = 1$ to $8$ do
        AssignParticleModified($\nu'_i$, $p$)
    else
        assign $p$ to $\nu$ and flag it as "outside"

if particle $p$ is inside $\nu$ then
    assign $p$ to $\nu$ and flag it as "inside"

The main difference is that we take the smoothing kernel size $h$ into account when assigning particles. The advantage is that this allows us to find neighboring particles very fast. The following algorithms show how we calculate the density of a particle before and after the modification.

Algorithm 6 Calculate Density

Require: A particle $p$

let $\rho$ be the density of particle $p$.
let $\nu$ be the node where $p$ is assigned to.
find $\nu$'s neighboring nodes, $\nu'_1, \ldots, \nu'_n$
for $i = 1$ to $n$ do
    let $p_1, \ldots, p_m$ be the particles of $\nu'_i$
    for $j = 1$ to $m$ do
        calculate the contribution of $p_j$ to $\rho$
With the modified octree structure, the algorithm will look like this:

**Algorithm 7 Calculate Density Modified**

**Require:** A particle $p$
- let $\rho$ be the density of particle $p$.
- let $\nu$ be the parent node of $p$.
- let $p_1 \ldots p_n$ be the particles assigned to $\nu$

**for** $i = 0$ to $n$ **do**
- calculate the contribution of $p_i$ to $\rho$

With these modifications, assigning particles to nodes and maintaining the tree is more time consuming. Fortunately, this is done only once per frame. Computing neighboring particles has to be done at least four times for each particle during the force calculation, eight more times for each node during the marching cubes algorithm and one more time for each node for identifying the different drops during the fluid trail synthesis. In practice, our modification to the octree data structure has made the simulation significantly faster.

### 4.4.3 Marching Cubes

Using a new data structure also had its effect on the marching cubes algorithm. Normally when using marching cubes the algorithm traverses a grid. To make a grid alongside the octree we were already using seemed redundant, so we slightly modified the marching cubes algorithm so it could traverse an octree. The simulation keeps a list of all the nodes in the octree that contain particles. The marching cubes algorithm only has to traverse the nodes that contain particles and create the mesh. As explained in Subsection 4.4.2, all the nodes that contain particles are the same size, which is preferred for the Marching Cubes algorithm to avoid incorrect results. Using a uniform grid in this case would also be a very expensive operation since, like we said before, there are likely to be a lot of empty grid cells.

We needed to adapt the standard octree to work better with the marching cubes algorithm. When we traverse a node in the octree we need to check the color field value at each point to see if it is inside or outside the fluid. To check the color field value at any point in space we need the density information of nearby particles. This requires finding and looking at the neighboring cells in the octree. Like we explained before, this is relatively expensive. Fortunately, we have solved this by modifying the octree structure, as explained in Subsection 4.4.2.

There is an additional problem specific to the marching cubes algorithm. A cell that does not contain particles could still be inside of the surface of the fluid. This is because the color field value is determined by the smoothing kernel of a particle. This is shown in Figure 4.15. If we only traverse the nodes that contain particles, the marching cubes algorithm will construct an incorrect surface of the color field. Traversing every node in the octree would also produce incorrect results and be redundant.
Figure 4.15: An example of particles contributing to the smoothed color field outside their nodes. Applying the marching cubes algorithm only to the nodes with particles would result in an incorrect mesh with holes in it. The blue dots are the particles, and are assigned to nodes at maximum depth. The blue areas are the smoothing kernels of the surface particles.

The alterations we discussed in the previous section helped speed up the calculations, but do not necessarily solve this problem. We need to create nodes at places that do not have particles but are within the influence of the smoothing kernels of the particles. For this we will show the algorithm to create new nodes. This is used at the start of the simulation to create the complete octree, and when new nodes need to be created to accommodate for particles which have moved outside their nodes.

**Algorithm 8 CreateNodes**

**Require:** A node $\nu$

1. if node $\nu$ is at maximum depth $d$ then
   return
2. if there are particles inside node $\nu$ then
   create 8 children ($\nu'_1, \ldots, \nu'_8$) for node $\nu$
   for $i = 1$ to 8 do
     CreateNodes($\nu'_i$)

This algorithm creates eight new children for a node if the current node has particles in it but is not yet at maximum depth. We need to vary this algorithm only slightly to create an octree that is suitable for the marching cubes algorithm to traverse. We want to have nodes we can use for our marching cube algorithm, and these need to cover all of the color field in order for the resulting surface to be complete. This means we may need to create more nodes that are within $h$ of a particle but do not contain particles themselves. So we modified the creation of nodes as shown below. This modification makes sure that we also keep dividing a node if there are particles within distance $h$ of it. The result is the creation of more nodes and ensures that the list of nodes used for the marching cubes algorithm is complete.
Algorithm 9 CreateNodes Modified

Require: A node $\nu$

if node $\nu$ is at maximum depth $d$ then
    return

Let $A$ be the area of node $\nu$ expanded by length $h$

if there are particles in the area $A$ then
    create 8 children ($\nu_1', \ldots, \nu_8'$) for node $\nu$

for $i = 1$ to $8$ do
    CreateNodes Modified($\nu_i'$)

4.5 Summary

In this chapter we have discussed every step of our fluid simulation engine. We have first discussed the physics behind our particle based fluid simulation and our addition to it. Next, we used marching cubes to visualise the fluid and added a way of simulating a fluid trail. Finally we discussed the implementation details involving our modified data structure. In the next chapter we will describe the interaction between facial animation and fluid simulation. We will combine tear generation with MPEG-4 Facial Animation, look at more ways to increase realism by using texture blending and discuss the parameters involved in our simulation.
Chapter 5

Integration

5.1 Introduction

In the previous chapters we explained how we animate the facial mesh and simulate the tears. Equally important is how we generate the tears and how the facial mesh and the simulated tears work together. In this chapter we will discuss our addition to the MPEG-4 Facial Animation standard in order to generate tears. In Section 5.3, we will discuss aspects of crying that cannot be visualised with facial animation or fluid simulation. In Section 5.4 we will discuss the various parameters we use and their effect in the real-time simulation. Finally, we will describe how the different parts of our application work together and show the user interface.

5.2 Tear generation

The MPEG-4 Facial Animation Standard is a simple and effective way of describing facial expressions, by defining a set of FAPs that control different logical areas of the face. Through these FAPs, an animator can quickly generate facial expressions without having to think about the geometrical deformations of the face mesh. The goal of our approach is to be able to have a similar way of controlling tears, but we do not want the animator to have to think of the physical behavior of tears.

To show that this is possible, we added two additional FAP parameters, one for each eye. The value of each FAP represents the number of particles that are present at that frame in the animation. This means that the FAP for generating tears is an absolute value that can be read independently from previous frames, just like the other FAPs. This way, the animator has complete control over the lifespan and the number of particles that are generated by the crying engine, for both left and right eyes. When the parameter that determines the number of particles decreases, particles have to be removed. We decided to remove the oldest particles since this seems the best way to represent tears drying up and disappearing. If we would remove the youngest particles, the newest tears would disappear, which looks strange. In addition, the source of both tear generating FAPs can be set, so that the animator can choose from where on the face tears are generated.

Another option for generating tears would have been to let the parameter determine how much particles are created at that frame. This allows for a more intuitive control over the particles. However, this requires us to give the particles a lifespan, otherwise the particles would never go away. This causes control over the particles to be less precise. Furthermore, all the other FAPs represent an absolute value at a certain frame, so it seemed logical to have this
5.3 Texture Blending

In section 2.2 we discussed several phenomena of crying. The most important ones, facial expressions and tears, were discussed in Chapter 3 and Chapter 4. Other phenomena, like blushing or red eyes can not be simulated by facial animation or fluid simulation. Although less important than facial expression or tears, we believe that adding these phenomena improves realism. Blushing, irritation of the skin or watery eyes are all small effects that make a simulation more realistic, and can be added to our crying simulation by using texture blending.

Texture blending uses a blending function to apply one texture over the other. This function determines how much is visible of the original texture and how much is visible of the overlay texture. Typically, these functions use an alpha value to determine the visibility of the overlay texture. We adjust this alpha value during the animation to make it correspond with the number of tears that are present, for example.

In our simulation we have implemented two of these textures. One for the skin around the eyes, and one for the eyes themselves. The overlay texture around the eyes is a copy of the skin, but made slightly darker and more red. We set the alpha value of this texture at zero at the start of the animation, and we want it to be one at the end of it. So we first look at the length of the animation and at each frame increase the alpha value of the overlay texture so that it is one at the end of the animation. This subtle detail represents the skin around the eyes getting wet.

The texture used for the eye itself is a partially transparent texture. We use the same material we use on the tears themselves, but we set its alpha value to zero at the start of the animation, so the eyes do not look watery at the start. We then increase the alpha value dependent on the number of particles present at the current frame. This causes the eyes to look shiny and reflective, as if they are wet.

Both these texture blends are not an exact representation of what happens when someone cries and are not linked to our tear generation in a physically correct way, but they add little details that makes the crying simulation seem more realistic. Examples of the texture blending can be seen in Chapter 6.

5.4 Physical Parameters

Because we combine facial animation and fluid simulation, a lot of different physical parameters are involved. Part of the parameters only apply to the fluid, but others define the interaction of the fluid with the skin. In this section, we give an overview of these parameters and how they affect the simulation.

First, we have to provide the three main parameters for the three forces used in our implementation of SPH. These are the pressure, the viscosity and the surface tension. The pressure determines how wide the particles of the tear will spread. Using a high pressure will result in large tears with low density particles. The viscosity will determine how well the particles drag each other along. Using a higher viscosity means that the tear will likely stay as one drop. The surface tension will also keep the particles together, but is less sensitive to obstacles and friction than the viscosity force.

Second, we have parameters for the external forces. These are skin adhesion, friction and gravity. A high skin adhesion force means that the tear will follow the surface of the face better.
and will be more affected by friction. The friction parameter itself is the most direct way of controlling how fast a tear moves across the skin. The influence of the gravity parameter is evident.

Next to the parameters related to forces, the size of the smoothing kernel of a particle can be set. A lower value for this will result in a more incoherent particle system because particles have less influence on each other. This value also influences the color field, so a smaller value means that the mesh will follow the actual shape of the particles more closely.

The previously discussed parameters are all used for SPH, but the mesh generation system also introduces some parameters. First, the node size determines how small the octree nodes containing the particles will be. A smaller node size results in more nodes but this can speed up the force calculations if the kernel size is small.

Second, the marching cube size needs to be set to a value smaller than or equal to the octree node size. A smaller marching cube size results in a more detailed mesh.

Finally, we can set the iso-value for the marching cubes algorithm. A higher iso-value results in a smaller mesh, following the particles more closely. The tear trails are generated automatically and they do not introduce additional parameters. In the next chapter, we will propose a few sample settings of these parameters that result in realistic tear simulation.

5.5 User Interface

In this chapter and the previous chapters we have discussed all the different parts of our facial animation engine. We have combined these parts of the simulation in one application. In Chapter 3 we have shown the application for creating facial expressions and animations. The application shown in this section is used for playing these facial animations and generating tears. A schematic overview of the application can be seen in Figure 5.1

![Schematic Overview](image)

Figure 5.1: A schematic overview of our application.
We will clarify the different segments of our application shown in Figure 5.1.

The main function creates a the SPH module, the Facial Animation engine, the FAP loader, the Physics engine and a frame listener at the start of the application. The FAP loader reads FAP files and stores the FAP values and frames.

The Facial animation engine creates a facial model, and updates this every frame with information from the FAP loader. It also updates the textures on the facial model, for texture blending effects. The SPH module initialises the particle system and the octree structure. Every frame it calculates the forces that are passed on to the Physics engine. The Physics engine applies the forces it receives from the SPH module to the particles. It also handles the collisions between the particles and the facial model.

The frame listener tells the Facial Animation engine and the Physics engine to update the face and particles. It also starts the Mesh Generation every frame. The Mesh Generation creates the meshes for the teardrops and the trails, using the particle information from the SPH module. It passes the mesh information on to the render engine. The Render engine renders the mesh provided by the Mesh Generation and the updated facial model provided by the Facial Animation Engine.

The application itself is quite straightforward. It shows a facial mesh which can be examined from all sides through intuitive camera controls. There are three buttons on the bottom left called "Load", "Play" and "Reset". The "Load" button allows a user to load a FAP file that contains the animation for the face. When the user presses "Play", the animation will start playing, including the production of tears, if defined in the animation. The "Reset" button causes the face to go back to its neutral pose and will destroy any tears that are present. The application with its user interface can be seen in Figure 5.2.

![Image](image-url)

Figure 5.2: The user interface of our application.
5.6 Summary

In this chapter, we gave an overview of the integration of facial animation and fluid simulation. We explained how we control the generation of tears by expanding the facial animation standard that we use in our simulation. This addition also lets us control other phenomena of crying, such as blushing or red eyes, as we discussed in Section 5.3. In Section 5.4 we described the parameters that are needed when combining fluid simulation and facial animation. Figure 5.3 shows a schematic overview of our simulation.

Figure 5.3: A schematic overview of our simulation. This includes the generation of tears, the animation of the mesh, the construction of the trail and blending of the facial texture.
Chapter 6

Results

6.1 Introduction

In the previous chapters we discussed the techniques we used for our simulation and we explained the implementation details of those techniques. In this chapter we show how these techniques affect our simulation. We will discuss the visual impact of the different aspects of the simulation and evaluate their performance. Also, we will show the effect of the different parameters we have discussed in the previous chapter. In the next section, we start by discussing the general performance of the simulation. In Sections 6.3 to 6.6 we will show the effect the SPH forces have on the fluid simulation. In Section 6.7 we will discuss the visual impact of adding more particles and the effect it has on the frame rate. The visual effects of trail synthesis and texture blending are shown in Sections ?? and ?? . Finally, we show the impact on the performance of different parts of the simulation in Section 6.10.

6.2 General Performance

We ran our simulation on a Pentium 4 Duo CPU 2.66 Ghz with 4 Gb RAM. We generated tears consisting of 100 particles and achieved an average frame rate of 12.7 fps. We have shown an example of our crying simulation in Figure 6.1, where we have combined facial expression with our modified fluid simulation. We also added the blending textures that provide some subtle details to the simulation. A few frames of a of a tear in our crying simulation can be seen in Figure 6.2. For these tests we used a pressure of $0.4e^{-2}$, a viscosity of 0.9, a tension of 0.2 and an adhesion of 20. The friction of the skin was set at 3.8. The size of the marching cubes value was half that of the node size. The iso-value was set to 0.02. We set the kernel size to 0.5. The meaning of these parameters is discussed in the previous chapter, in Section 5.4. The values of these parameters are mostly found by trial and error, since in our application, the real physical properties of water did not produce satisfying results. This may be because we have a small number of particles or because the weight of the particles is different from the weight of amount of water they represent. For each eye, a particle source is defined. The position of the sources is set at the middle of each eye. In the next sections we will look at several different aspects and parameters of the fluid simulation and their effect on the simulation.

44
6.3 Pressure

We will start with discussing the influence of the pressure parameter. The pressure determines how wide the particles of the tear will spread. Using a high pressure will result in large tears with low density particles. It does not affect frame rate unless the pressure is so high that the fluid will explode. This causes a lot of particles to be spread over a large area, resulting in a lower frame rate. Figure 6.3 will show the influence of pressure. In these pictures we have disabled the fluid trail synthesis to increase the visibility of the tear itself. We also shown the particles that form the fluid, so the particle distances can clearly be seen.
6.3 On the left we see a tear with a pressure of $0.5e^{-2}$. This is a good setting for our simulation. Increase of pressure can be seen in the middle, where the pressure is set to $0.5e^{-1}$. It clearly shows that the particles are pushed in every direction, generating a large volume of water. On the right a tear with a pressure of $0.5e^{-3}$ is shown. Less particles leave the eyelid because they are not pushed by the pressure. The particles in the teardrop are very close together.

6.4 Viscosity

The next parameter is viscosity. The viscosity will determine how well the particles drag each other along. Using a higher viscosity means that the tear will likely stay as one drop. Using a higher viscosity means that a moving particle will drag others along, creating larger drops. Since viscosity causes particles to match the velocity of their surrounding particles, using a high viscosity will result in a very rigid teardrop, since all the particles in the drop will be moving more or less in the same direction. Figure 6.4 shows the results of varying viscosity values. Again, we disabled the fluid trail and made the particles visible.

Figure 6.4: The teardrop on the left has a viscosity of 0.8 which produces a nice result. There are several independent drops of a normal size. In the middle a teardrop with viscosity 1.4 is shown. A lot of particles from the eyelid are drawn into one drop. This is still a pretty good result if larger drops are desired, but the motions will be very rigid. On the left we show tears with a viscosity of 0.2. Particles that are falling down do not drag any particles along with them, resulting in teardrops consisting of only one particle. This is undesirable since the motions of these drops are very unrealistic.
6.5 Surface Tension

Surface tension is the force that holds stabilizes the fluids surface. Since the teardrops consist of a small number of particles, compared to the water recreated in [18], it has an effect on all the particles in a drop and cause them to stay together. This is also done by the viscosity, but the surface tension is less influenced by obstacles and friction. Setting a high surface tension causes the particles to clutter closely together, creating smaller drop. A low surface tension results in an unstable fluid. This can be seen in Figure 6.5.

Figure 6.5: The example on the left has a surface tension of 0.2. A sufficient number of particles moved down from the eyelid to create nice little clusters of tears. The surface tension of the example in the middle is set to 0.0. The particles are easily dragged down because of the viscosity and apart from that, nothing keeps them together. On the right we show the result of a surface tension of 1.0, which is most likely too high. Too few particles are dragged down, and the few that are stick very close together, creating very small round droplets.

6.6 Adhesion

The adhesion is a force we added to the SPH-method to make sure that the fluid stick to the surface. It is pulling the particles into the skin, in the direction of the normal of the skin. Without or with too little adhesion, the particles would just slide off the skin as soon as they passed the upper half of the cheek. Setting the adhesion too high may cause problems with the physics engine, depending on the precision of its calculations. This may result in the particle being launched in seemingly random directions. Higher adhesion values also cause the friction to have more effect, because the particle is being pulled firmly into the surface. The results are shown in Figure 6.6.

6.7 Particle Count

In the previous examples we have used 100 particles for our tears. In this section we are going to look at the effects of the number of particles we generate. This has consequences for the frame rate of our simulation and the appearance of the tears. When using less particles we can also set the pressure higher to increase the volume of the fluid. Here we enable the fluid trail synthesis and do not show the particles, so we have a better impression of how the tears look. We first generated tears using 100 particles using the parameters we have just determined to be desirable. We then used the same parameters to generate tears using 50 particles, but we
Figure 6.6: These examples show the influence of adhesion. On the left, we have a normal adhesion set at 20. The tear in this example rolls down quite far, to show that it stays on the surface. In the example in the middle we have set the adhesion to 100. The collision handling with forces this big is not precise enough, so the particles are being pulled through the mesh or repulsed in seemingly random directions. On the right we have turned the adhesion off. The result is that the particles just fall down after the surface of the skin points downwards.

doubled the pressure to simulate more or less the same volume. We then did a third test using only 20 particles and again doubling the pressure. The results can be seen in Figure 6.8. The frame rates achieved with these tests are shown in Figure 6.7.

Figure 6.7: This graph shows the achieved frame rates with the corresponding number of particles used.
Figure 6.8: From left to right are the results of tears generated with respectively 100, 50 and 20 particles. The results are remarkably similar, but the tears with more particles appear more voluminous.

6.8 Trail Synthesis

The fluid trail synthesis method, which we explained in Chapter 4, is an addition to the fluid simulation. It simulates the trail a fluid leaves when is traverses a surface. Our goal was to make it a fast technique to visually enhance the simulation. We generated two sets of tears, one just uses the basic fluid simulation, and the other uses the fluid trail synthesis method to calculate and visualise the trail of the tear. Both sets of tears had the same parameters as described in our General Results section and used 100 particles. Like in our general results section, when enabling the trail we got a frame rate of 12.7 frames per second, so using $\approx 0.079$ seconds per frame. Disabling the trail increased the frame rate to 13.5 fps, meaning $\approx 0.074$ seconds per frame. So our fluid trail synthesis uses $\approx 0.005$ seconds per frame. The visual contribution of the technique can be seen in Figure 6.9.

Figure 6.9: This picture shows the difference trail synthesis makes to the crying simulation. The example on the left has trail synthesis disabled and it is enabled on the right.
6.9 Texture Blending

In the chapter "Integration" we described techniques for using blending textures to enhance the realism of our crying simulation. These are very easy to implement techniques and cost very little CPU time. In Figure 6.10 we show you the results when adding texture blending.

Figure 6.10: In this picture, the addition of texture blending can be seen. On the left, we do not blend the skin or eyes with other textures, and on the left we applied texture blending to both.

6.10 Frame Rate Measurements

Finally, to get a better idea of the computation time needed for each aspect of our simulation, we are going to measure the frame rates with several features enabled and disabled. To get a good idea of the scalability of our simulation, we have measured the frame rate these features using both 100 and 50 particles. The resulting frame rates can be seen in Figure 6.11 and Table 6.1. First, we disabled everything in our simulation. This gave us an idea of how fast the simulation was with just the particles, with just the physics engine testing for collisions. We noticed a big drop in frame rate when enabling the octree. Enabling the density calculations and the force calculations further lowered the frame rate substantially. Finally we enabled the marching cubes algorithm and the trail synthesis for the complete simulation. Notable is that the increase in particles seem to have a big effect on the octree maintenance and the density calculations. For example, when enabling the octree, when using 100 particles, the time used per frame is $\approx 0.008$ seconds. The time used per frame when then enabling the density calculations is $\approx 0.020$. That is a difference of $\approx 0.012$ seconds. When using 50 particles, this difference is $\approx 0.0012$. So it costs about ten times as much when using twice as much particles. Remarkable is the effect on the force calculations. When enabling them, the simulation needs $\approx 0.052$ seconds per frame when using 100 particles and $\approx 0.022$ seconds per frame when using 50 particles. This means that computing the forces costs $\approx 0.032$ seconds when using 100 particles, and $\approx 0.019$ when using 50 particles. This is less than twice the time to calculate the force for twice as much particles. We think that this is because of the addition of the forces, the particles will be spread out more over the octree, which means there will be less particles per node. This has relatively more effect on 100 particles than it has on 50 particles. This also explains why there is almost no difference between the computation times needed for the marching cubes algorithm.
Figure 6.11: In this bar graph we show the difference in frame rate between 100 and 50 particles while disabling several parts of our simulation. The parts of the simulation listed at each column work cumulative. So, the column labelled "octree" means that we only maintain the octree. The column labelled "density" means we have turned on the octree maintenance and the density calculations.

<table>
<thead>
<tr>
<th></th>
<th>Nothing</th>
<th>Octree</th>
<th>Density</th>
<th>Forces</th>
<th>Marching Cubes</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 Particles</td>
<td>400 fps</td>
<td>121 fps</td>
<td>49 fps</td>
<td>19 fps</td>
<td>13 fps</td>
</tr>
<tr>
<td>50 Particles</td>
<td>1100 fps</td>
<td>454 fps</td>
<td>293 fps</td>
<td>45 fps</td>
<td>37 fps</td>
</tr>
</tbody>
</table>

Table 6.1: The frame rates that are shown in the graph in Figure 6.11

6.11 Summary

In this chapter we have discussed the results of various aspects of our simulation. We have shown the effect of the different forces used in the SPH method. We have indicated that reducing the number of particles can be useful in our simulation. We have shown the visual effect of the addition of trail synthesis and texture blending. Finally, we have measured the frame rates of different parts of our simulation. In the next chapter, we will draw our conclusions from these results.
Chapter 7

Conclusions and Future Work

7.1 Introduction

In the previous chapters we have proposed our technique and discussed our results. In this chapter, in Section 7.2, we will first present the conclusions of this thesis project, based on the results from Chapter 6. Second, we will discuss possible future work that can be done for crying simulations in Section 7.3.

7.2 Conclusions

We have presented a real-time crying simulation framework, by using an extended SPH approach, optimised for crying fluid simulation. Our framework integrates with an existing facial animation system, and it is independent of the renderer and physics engine that we used. The shape of the fluid and the material used create a convincing simulation of tears. By adding a skin adhesion force, the motion of the tear neatly follows the skin without falling off.

We added a technique to synthesise a fluid trail, making a great visual contribution to the simulation while being computationally inexpensive. The effectiveness of the addition of texture blending can be argued, but in our opinion, the subtle detail they provide is an improvement to the simulation and put virtually no strain on the CPU.

Because of the easy integration with existing facial animation frameworks, animators will be able to control crying motions using only a few parameters in addition to the FAP values as defined in the MPEG-4 standard. For more control, more detailed parameters can be altered, such as the viscosity of the fluid or the adhesion of the skin. As shown in Chapter 6, these parameters have an effect on the fluid simulation resembling the effect they have on the behavior of a fluid.

The frame rates we achieved are shown in Chapter 6 and show that the framework can be used for real-time crying, although currently it is too slow to be implemented in games or other interactive simulations. In conclusion, we think that this framework can be a good addition to any simulation involving user interaction with virtual characters, given that the frame rate can be improved further.
7.3 Future Work

Although we have set the first steps in the direction of creating a more expressive face, a lot of work can still be done.

Extreme emotions such as laughing and crying are difficult to simulate convincingly without properly modeling muscle motions. In the future, we want to look at what muscle motions are important in crying and laughing animations in order to provide more detailed facial animations, while retaining the real-time constraint.

Another important consideration is the control of the facial expression and tears. As discussed in Chapter 2, several motivations and emotions are related with crying. An interesting research objective would be to investigate how such motivations and emotions can be incorporated in embodied conversational agents. This could result in an implementation where tears and crying motions may be controlled automatically by describing emotions rather than expressions.

Other applications for our simulation of small scale fluids can also be explored. We imagine it can also be used for rain or sweat for example. This has not only an effect on the realism of expressions, but can really contribute to the ambiance of a scene or setting. This makes our proposed method not only interesting for current facial animation engines but for a larger range of applications as well.

Although the fluid simulation meets the real-time constraint while producing tears that look satisfying, we noticed in Chapter 6 that it has a big impact on the frame rate. Especially when we need to maintain the modified octree data structure and when we add forces, we see the speed drop significantly. Since our decision for the modified octree data structure was based on fluid simulation for larger fluid volumes, we think there may be better options available for this specific situation. Also, when testing various settings for the adhesion parameter in Chapter 6, we have noticed problems with the accuracy of the physics engine. This accuracy can be set higher, but this will effect the frame rate even further. This leads us to believe that research into a more suitable physics engine could be valuable for this framework.
Bibliography


## Appendix A

### FAP Definition Table

This appendix shows the table with the MPEG-4 Facial Animation FAP definitions.

<table>
<thead>
<tr>
<th>nr.</th>
<th>FAP name</th>
<th>FAP description</th>
<th>Units</th>
<th>Uni / Bidir</th>
<th>Motion</th>
<th>Grp</th>
<th>Subgrp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Visime</td>
<td>Set of values determining the mixture of two visemes for this frame</td>
<td>na</td>
<td>na</td>
<td>na</td>
<td>1</td>
<td>na</td>
</tr>
<tr>
<td>2</td>
<td>Expression</td>
<td>A set of values determining the mixture of two facial expression</td>
<td>na</td>
<td>na</td>
<td>na</td>
<td>1</td>
<td>na</td>
</tr>
<tr>
<td>3</td>
<td>open_jaw</td>
<td>Vertical jaw displacement (does not affect mouth opening)</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>lower_t_midlip</td>
<td>Vertical top middle inner lip displacement</td>
<td>MNS</td>
<td>B</td>
<td>down</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>raise_b_midlip</td>
<td>Vertical bottom middle inner lip displacement</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>stretch_l_cornerlip</td>
<td>Horizontal displacement of left inner lip corner</td>
<td>MW</td>
<td>B</td>
<td>left</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>stretch_r_cornerlip</td>
<td>Horizontal displacement of right inner lip corner</td>
<td>MW</td>
<td>B</td>
<td>right</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>lower_t_lip_lm</td>
<td>Vertical displacement of midpoint between left corner and middle of top inner lip</td>
<td>MNS</td>
<td>B</td>
<td>down</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>lower_t_lip_rm</td>
<td>Vertical displacement of midpoint between right corner and middle of top inner lip</td>
<td>MNS</td>
<td>B</td>
<td>down</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>nr.</td>
<td>FAP name</td>
<td>FAP description</td>
<td>Units</td>
<td>Uni / Bidir</td>
<td>Motion</td>
<td>Grp</td>
<td>Subgrp</td>
</tr>
<tr>
<td>-----</td>
<td>----------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>-------</td>
<td>-------------</td>
<td>--------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>10</td>
<td>raise_b_lip_lm</td>
<td>Vertical displacement of midpoint between left corner and middle of bottom inner lip</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>raise_b_lip_rm</td>
<td>Vertical displacement of midpoint between right corner and middle of bottom inner lip</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>raise_l_cornerlip</td>
<td>Vertical displacement of left inner lip corner</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>raise_r_cornerlip</td>
<td>Vertical displacement of right inner lip corner</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>thrust_jaw</td>
<td>Depth displacement of jaw</td>
<td>MNS</td>
<td>U</td>
<td>forward</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>shift_jaw</td>
<td>Side to side displacement of jaw</td>
<td>MW</td>
<td>B</td>
<td>right</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>push_b_lip</td>
<td>Depth displacement of bottom middle lip</td>
<td>MNS</td>
<td>B</td>
<td>forward</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>push_t_lip</td>
<td>Depth displacement of top middle lip</td>
<td>MNS</td>
<td>B</td>
<td>forward</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>depress_chin</td>
<td>Upward and compressing movement of the chin</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>19</td>
<td>close_t_l_eye</td>
<td>Vertical displacement of top left eyelid</td>
<td>IRISD</td>
<td>B</td>
<td>down</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>close_t_r_eye</td>
<td>Vertical displacement of top right eyelid</td>
<td>IRISD</td>
<td>B</td>
<td>down</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>close_b_l Eye</td>
<td>Vertical displacement of bottom left eyelid</td>
<td>IRISD</td>
<td>B</td>
<td>up</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>close_b_r_eye</td>
<td>Vertical displacement of bottom right eyelid</td>
<td>IRISD</td>
<td>B</td>
<td>up</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>yaw_l_eye</td>
<td>Horizontal orientation of left eyeball</td>
<td>AU</td>
<td>B</td>
<td>left</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>yaw_r_eye</td>
<td>Horizontal orientation of right eyeball</td>
<td>AU</td>
<td>B</td>
<td>left</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>25</td>
<td>pitch_l_eye</td>
<td>Vertical orientation of left eyeball</td>
<td>AU</td>
<td>B</td>
<td>down</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>pitch_r_eye</td>
<td>Vertical orientation of right eyeball</td>
<td>AU</td>
<td>B</td>
<td>down</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>27</td>
<td>thrust_l_eye</td>
<td>Depth displacement of left eyeball</td>
<td>ES</td>
<td>B</td>
<td>forward</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td>thrust_r_eye</td>
<td>Depth displacement of right eyeball</td>
<td>ES</td>
<td>B</td>
<td>forward</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>29</td>
<td>dilate_l_pupil</td>
<td>Dilation of left pupil</td>
<td>IRISD</td>
<td>B</td>
<td>growing</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>dilate_r_pupil</td>
<td>Dilation of right pupil</td>
<td>IRISD</td>
<td>B</td>
<td>growing</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>nr.</td>
<td>FAP name</td>
<td>FAP description</td>
<td>Units</td>
<td>Uni / Bidir</td>
<td>Motion</td>
<td>Grp</td>
<td>Subgrp</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------</td>
<td>------------------------------------------</td>
<td>-------</td>
<td>-------------</td>
<td>--------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>31</td>
<td>raise_l_i_eyebrow</td>
<td>Vertical displacement of left inner eyebrow</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>raise_r_i_eyebrow</td>
<td>Vertical displacement of right inner eyebrow</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>raise_l_m_eyebrow</td>
<td>Vertical displacement of left middle eyebrow</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>34</td>
<td>raise_r_m_eyebrow</td>
<td>Vertical displacement of right middle eyebrow</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>35</td>
<td>raise_l_o_eyebrow</td>
<td>Vertical displacement of left outer eyebrow</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>36</td>
<td>raise_r_o_eyebrow</td>
<td>Vertical displacement of right outer eyebrow</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>37</td>
<td>squeeze_l_eyebrow</td>
<td>Horizontal displacement of left eyebrow</td>
<td>ES</td>
<td>B</td>
<td>right</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>squeeze_r_eyebrow</td>
<td>Horizontal displacement of right eyebrow</td>
<td>ES</td>
<td>B</td>
<td>left</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>39</td>
<td>puff_l_cheek</td>
<td>Horizontal displacement of left cheek</td>
<td>ES</td>
<td>B</td>
<td>left</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>puff_r_cheek</td>
<td>Horizontal displacement of right cheek</td>
<td>ES</td>
<td>B</td>
<td>right</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>41</td>
<td>lift_l_cheek</td>
<td>Vertical displacement of left cheek</td>
<td>ENS</td>
<td>U</td>
<td>up</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>42</td>
<td>lift_r_cheek</td>
<td>Vertical displacement of right cheek</td>
<td>ENS</td>
<td>U</td>
<td>up</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>43</td>
<td>shift_tongue_tip</td>
<td>Horizontal displacement of tongue tip</td>
<td>MW</td>
<td>B</td>
<td>right</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>raise_tongue_tip</td>
<td>Vertical displacement of tongue tip</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>thrust_tongue_tip</td>
<td>Depth displacement of tongue tip</td>
<td>MW</td>
<td>B</td>
<td>forward</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>raise_tongue</td>
<td>Vertical displacement of tongue</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>47</td>
<td>tongue_roll</td>
<td>Rolling of the tongue into U shape</td>
<td>AU</td>
<td>U</td>
<td>concave, upward</td>
<td>6</td>
<td>3, 4</td>
</tr>
<tr>
<td>48</td>
<td>head_pitch</td>
<td>Head pitch angle from top of spine</td>
<td>AU</td>
<td>B</td>
<td>down</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>head_yaw</td>
<td>Head yaw angle from top of spine</td>
<td>AU</td>
<td>B</td>
<td>left</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>head_roll</td>
<td>Head roll angle from top of spine</td>
<td>AU</td>
<td>B</td>
<td>right</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>51</td>
<td>lower_t_midlip_o</td>
<td>Vertical top middle outer lip displacement</td>
<td>MNS</td>
<td>B</td>
<td>down</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>52</td>
<td>raise_b_midlip_o</td>
<td>Vertical bottom middle outer lip displacement</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>nr.</td>
<td>FAP name</td>
<td>FAP description</td>
<td>Units</td>
<td>Uni / Bidir</td>
<td>Motion</td>
<td>Grp</td>
<td>Subgrp</td>
</tr>
<tr>
<td>-----</td>
<td>------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>-------</td>
<td>-------------</td>
<td>--------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>53</td>
<td>stretch_l_cornerlip_o</td>
<td>Horizontal displacement of left outer lip corner</td>
<td>MW</td>
<td>B</td>
<td>left</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>54</td>
<td>stretch_r_cornerlip_o</td>
<td>Horizontal displacement of right outer lip corner</td>
<td>MW</td>
<td>B</td>
<td>right</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>55</td>
<td>lower_l_lip_lm_o</td>
<td>Vertical displacement of midpoint between left corner and middle of top outer lip</td>
<td>MNS</td>
<td>B</td>
<td>down</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>56</td>
<td>lower_l_lip_rm_o</td>
<td>Vertical displacement of midpoint between right corner and middle of top outer lip</td>
<td>MNS</td>
<td>B</td>
<td>down</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>57</td>
<td>raise_b_lip_lm_o</td>
<td>Vertical displacement of midpoint between left corner and middle of bottom outer lip</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>58</td>
<td>raise_b_lip_rm_o</td>
<td>Vertical displacement of midpoint between right corner and middle of bottom outer lip</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>59</td>
<td>raise_l_cornerlip_o</td>
<td>Vertical displacement of left outer lip corner</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>60</td>
<td>raise_r_cornerlip_o</td>
<td>Vertical displacement of right outer lip corner</td>
<td>MNS</td>
<td>B</td>
<td>up</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>61</td>
<td>stretch_l_nose</td>
<td>Horizontal displacement of left side of nose</td>
<td>ENS</td>
<td>B</td>
<td>left</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>62</td>
<td>stretch_r_nose</td>
<td>Horizontal displacement of right side of nose</td>
<td>ENS</td>
<td>B</td>
<td>right</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>63</td>
<td>raise_nose</td>
<td>Vertical displacement of nose tip</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>64</td>
<td>bend_nose</td>
<td>Horizontal displacement of nose tip</td>
<td>ENS</td>
<td>B</td>
<td>right</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>65</td>
<td>raise_l_ear</td>
<td>Vertical displacement of left ear</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>66</td>
<td>raise_r_ear</td>
<td>Vertical displacement of right ear</td>
<td>ENS</td>
<td>B</td>
<td>up</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>67</td>
<td>pull_l_ear</td>
<td>Horizontal displacement of left ear</td>
<td>ENS</td>
<td>B</td>
<td>left</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>68</td>
<td>pull_r_ear</td>
<td>Horizontal displacement of right ear</td>
<td>ENS</td>
<td>B</td>
<td>right</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

Table A.1: The table that contains the FAP definitions.
Appendix B

Simulation Parameters

This appendix lists all the parameters we use in our simulation. We listed ideal values for each parameter that provide satisfying results together. These ideal values were found by trial and error and are very dependent of each other.

- **gravity**
  Sets direction and magnitude of the gravity.
  Ideal value: 0, -9.81, 0

- **h**
  The smoothing kernel size. Its ideal value is dependent on the scale of the simulation.
  Ideal value: 0.5

- **cfkernel**
  A different smoothing kernel size that can be used for calculating the color field only.
  Ideal value: 0.5

- **gridsize**
  The size of the bottom nodes of the octree. Determines the maximum depth of the octree.
  Ideal value: 0.5

- **cubysize**
  The size of the cubes used for the marching cubes algorithm. Must be the gridsize divided by an integer.
  Ideal value: 0.25

- **friction**
  The friction that is set at contact between a particle and the skin. Ideal value depends on the mass of the particles.
  Ideal value: 3.8

- **bounciness**
  The bounciness that can be set at contact between a particle and the skin. Our simulation does not use bouncyness.
  Ideal value: 0.0

- **k**
  The pressure constant.
  Ideal value: 0.4e-2
• tension
  The surface tension constant.
  Ideal value: 0.2;

• mu
  The viscosity constant.
  Ideal value: 0.9

• alpha
  The constant used for the adhesion to the skin.
  Ideal value: 15.0;

• cubemass
  The mass of the body of the particle used by the physics engine
  Ideal value: 1

• particlesize
  The size of the body of the particle used by the physics engine
  Ideal value: 0.1

• particlemass
  The mass of the particle used by the SPH calculations. This can be set different from the
  mass of its physical body
  Ideal value: 1

• isovalue
  The iso-value used for determining the isosurface
  Ideal value: 0.02;

• traveldist
  The distance over which we gradually reduce the mass of the particles. Should be about
  as large as the half the height of the face.
  Ideal value: 10.0;

• showparticles
  A boolean that determines whether we show the particles or not.
  Ideal value: false;